ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικό και Καποδιστριακό
Πανεπιστήμιο Αθηνών

Ανάπτυξη Λογισμικού σε Πληροφοριακά Συστήματα

Ευαγγελινού Ευστρατία - 1115201500038
Καζαντζίδου Σοφία - 1115201500051
Λίτσος Λουκάς - 1115201500082

21 Ιανουαρίου 2020

Περιεχόμενα

Εισαγωγή

Το πρόγραμμα δέχεται για είσοδο ένα σύνολο αρχείων, από τα οποία το καθένα περιέχει τα δεδομένα ενός πίνακα(αριθμούς). Στη συνέχεια, δέχεται ερωτήματα σε ψευδο-SQL γλώσσα. Για να παράξει τα αποτελέσματα χρησιμοποιεί την τεχνική της $merge-join$. Ταξινομεί τους πίνακες, ανά κομμάτια, και στην συνέχεια συγκρίνει τις εμπλεκόμενες στήλες του ερωτήματος με πολυπλοκότητα $O(n)$. Βρίσκει τα αποτέλεσμα στο ερώτημα και εκτυπώνει το άθροισμα των γραμμών του αποτελέσματος για κάθε ζητούμενο.

1 Υλοποίηση

Αρχικά,με τη χρήση της $mmap$ αποθηκεύονται σε μία κλάση όλα τα δεδομένα των πινάκων. Στην σε μια άλλη κλάση αποθηκεύονται το ερώτημα. Σε αυτήν φτιάχνεται μία ουρά όπου υπάρχουν όλα τα $predicates$ με κατάλληλη σειρά(αναλυτικά παρακάτω).

Στην συνάρτηση $join$ γίνονται πρώτα τα φίλτρα και τα κρατάμε σε ενδιάμεσο αποτέλεσμα, πίνακα. Στην συνέχεια κάθε $join$ που γίνεται αν ο πίνακας υπάρχει στα ενδιάμεσα αποτελέσματα χρησιμοποιούμε τον πίνακα από εκεί αλλιώς παίρνουμε τον κανονικό πίνακα και κάνουμε το $join$, ταξινομώντας τον πίνακα ανά κομμάτια, ανάλογα με τα $bytes$.

Τέλος, Χρησιμοποιώντας το ιστόγραμμα, μία δομή που λειτουργεί ως ευρετήριο και βοηθάει στην άμεση προσπέλαση των δεδομένων του πίνακα, γίνονται οι τελικές συγκρίσεις και παράγεται το αποτέλεσμα.

2 Βελτιώσεις

Παρακάτω αναφέρονται όλες οι βελτιώσεις που έγιναν με σκοπό την μείωση του χρόνου εκτέλεσης του προγράμματος.

2.1 Διάφορες Βελτιώσεις 2ου μέρους

2.1.1 Αναδιοργάνωση Ουράς Προτεραιότητας

Αντί να έχουμε στην ουρά τα φίλτρα και έπειτα τις ζεύξεις με σχεδόν τυχαία σειρά, κάθε φορά που γινόταν ένα $predicate$, αλλάζαμε τη σειρά φέρνοντας μπροστά το $predicate$ που για τη δεδομένη κατάσταση ήταν το βέλτιστο. Η σειρά που ακολουθήσαμε ήταν να δίνουμε προτεραιότητα στα φίλτρα με αριθμό, μετά στα $predicate$ που οι πίνακες υπάρχουν στα ενδιάμεσα αποτελέσματα και άρα είναι έμμεσο φίλτρο, μετά στα $predicate$ με πίνακες που έχουν υποστεί και οι δύο φίλτρο και τέλος στα $predicate$ με τουλάχιστον έναν πίνακα που έχει υποστεί φίλτρο.

2.1.2 Αξιοποίηση Ταξινομημένου Πίνακα

Προσθέσαμε μία μέθοδο η οποία επιστέφει αν μία στήλη ενός πίνακα είναι ταξινομημένη. Αν είναι και χρησιμοποιείται ξανά σε ερώτημα τότε κάνουμε αυτό το ερώτημα χωρίς να ταξινομούμε ξανά τον πίνακα. Αυτή η αλλαγή βελτίωσε τον χρόνο εκτέλεσης με τα μεσαίου μεγέθους δεδομένα εισόδου έως και $20sec$.

2.2 Στατιστικά Ερωτημάτων

Σκοπός αυτής της βελτίωσης είναι να προβλέψει το πλήθος των ενδιάμεσων αποτελεσμάτων κάθε ζεύξης και να προτείνει τον συνδυασμό με το λιγότερο κόστος. Πρόκειται, δηλαδή για μία πρόβλεψη η οποία μπορεί και να διαφέρει από την πραγματικότητα. Πιο συγκεκριμένα, υπολογίσαμε ένα σύνολο στατιστικών για κάθε στήλη κάθε πίνακα. Βρήκαμε την μικρότερη και τη μεγαλύτερη τιμή, το πλήθος των δεδομένων και το πλήθος των μοναδικών δεδομένων. Η τεχνική και οι τύποι που χρησιμοποιήθηκαν για την πρόβλεψη των στατιστικών μπορούν να εφαρμοστούν μόνο για τα συγκεκριμένα δεδομένα και δεν μπορούν να χρησιμοποιηθούν υπό οποιαδήποτε συνθήκες. Τρέξαμε το πρόγραμμα με τη δημιουργία αυτών των στατιστικών και τον συγκρίναμε με τον χρόνο που κάνει το πρόγραμμα χωρίς αυτά, με την ουρά προτεραιότητας που δημιουργήσαμε.

Μεσαίου μεγέθους είσοδος

1. Με στατιστικά : 70.488 *sec*
2. Χωρίς στατιστικά : 55.260 *sec*

Μικρού μεγέθους είσοδος

1. Με στατιστικά : 0.355 *sec*
2. Χωρίς στατιστικά : 0.298 *sec*

Επίσης, μετρήσαμε τη μνήμη που καταναλώνει το πρόγραμμα αν του δοθεί ως είσοδος του *small dataset*, με και χωρίς στατιστικά.

1. Με στατιστικά : 271.945.785 *bytes*
2. Χωρίς στατιστικά : 271.522.672 *bytes*

Από τους παραπάνω χρόνους(μέσος όρος 5 εκτελέσεων) συμπεραίνουμε ότι τα στατιστικά δεν βελτίωσαν τον χρόνο εκτέλεσης του προγράμματος. Αυτό μπορεί να ευθύνεται στο ότι η ουρά προτεραιότητας που υλοποιήσαμε έκανε καλύτερες επιλογές για κάθε δεδομένη χρονική στιγμή ενώ τα στατιστικά δεν κατάφεραν να προβλέψουν το αληθινό κόστος. Ενώ πολύ πιθανό η διαφορά μνήμης κατά $423,113$ *bytes* παραπάνω με τα στατιστικά μπορεί να επηρεάζει αρνητικά την απόδοση τους.

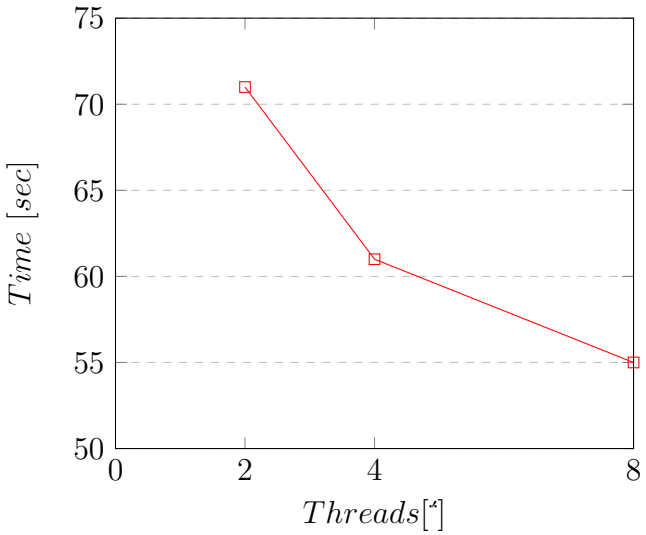
2.3 Παραλληλοποίηση

Σε αυτήν την ενότητα παραθέτουμε τον τρόπο που υλοποιείται η παραλληλοποίηση και κάποια διαγράμματα που συγκρίνουν τους χρόνους εκτέλεσης του προγράμματος σε σχέση με κάποιες μεταβλητές. Τα διαγράμματα με τη Κόκκινη γραμμή αφορούν την εκτέλεση με το μεσαίο σετ δεδομένων, ενώ αυτά με τη Μπλε γραμμή αφορούν αυτήν με το μικρό σετ δεδομένων. Οι χρόνοι που έχουν χρησιμοποιηθεί είναι μέσος όρος 5 εκτελέσεων με τις ίδιες παραμέτρους. Αυτό που παρατηρήσαμε και θα φανεί και απο τους παρακάτω χρόνους εκτέλεσης είναι ότι κάθε μορφή παραλληλοποίηση βοηθάει στην ελαχιστοποίηση του χρόνου εκτέλεσης.

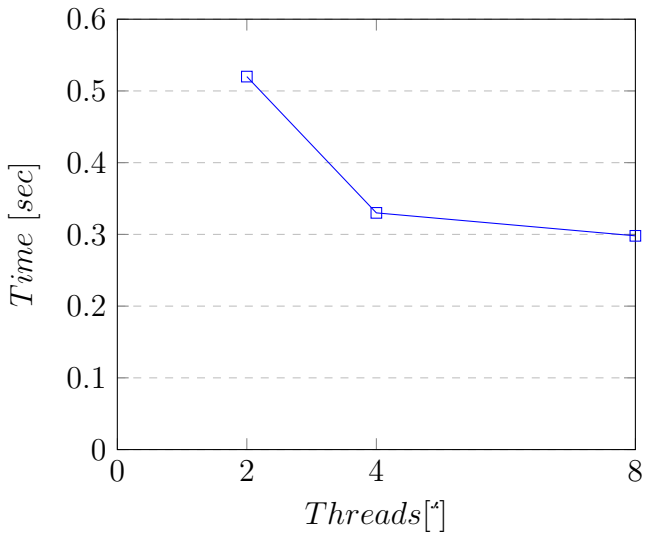
2.3.1 *Job Scheduler*

Το *API* της κλάσης *Job Scheduler* έχει υλοποιηθεί με βάση αυτό της εκφώνησης.Ο *Job Scheduler* χρησιμοποιεί όσα *threads* έχει το μηχάνημα στο οποίο εκτελειται το πρόγραμμα, εκτός αν έχει καθοριστεί συγκεκριμένο νούμερο μέσω της $defined$ μεταβλητής $THREADS$. Ανάλογα με το μέγεθος *thread pool* που αλλάζει και ο χρόνος εκτέλεσης του προγράμματος όπος φαίνεται στο παρακάτω διάγραμμα:

Για το *medium*:



Για το *small*:



Τα αποτελέσματα αυτά είναι με παραλληλοποίηση στο *Query*, στο *Sort* και στο *Join*.

2.3.2 Επιπέδου Επερώτησης (Queries)

Λεπτομέρειες Υλοποίησης Για την υλοποίηση αυτή κάθε φορά που εισάγει ο χρήστης ένα *query* γίνεται *schedule*, ενώ όταν εισάγει τον χαρακτήρα F χρησιμοποιείται ο *barrier* του *Job Scheduler* για να ολοκληρωθούν όλα τα ερωτήματα του *batch* και να γίνουν $flush$ τα αποτέσματα.

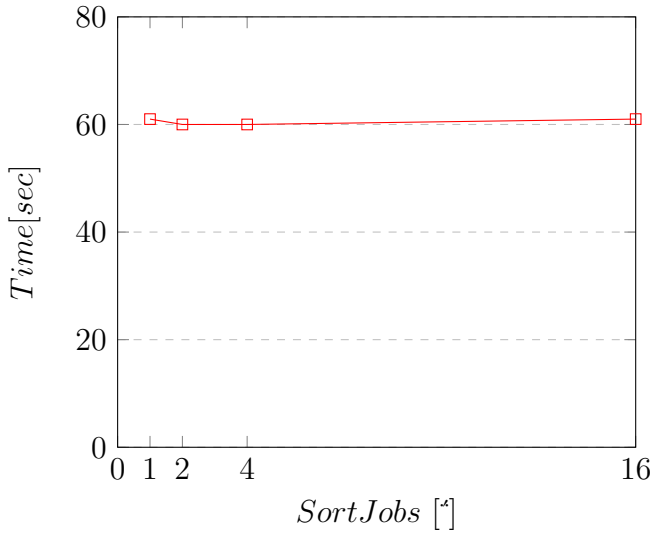
Χρόνοι Υλοποίησης Χωρίς παραλληλοποίηση σε άλλα σημεία του προγράμματος, όταν προστεθεται η παραλληλία στο *query* ο χρόνος του *medium* υποδιπλασιάζεται, ενώ ο χρόνος του *small* έχει μια μικρή βελτίωση.
Όταν προσθέτουμε παραλληλία και σε άλλα σημεία του προγράμματος πάλι η παραλληλοποίηση στο *query* μειώνει κι άλλο το χρόνο αλλα οι διαφορές είναι μικρότερες.

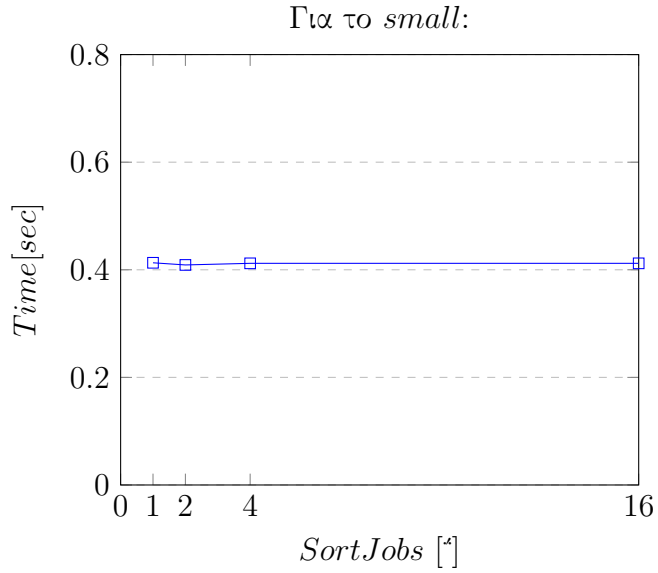
2.3.3 Επιπέδου Ταξινόμησης (Sort)

Λεπτομέρειες Υλοποίησης Για τηνυλοποίση αυτής της παραλλοποίησης το *job* που εκτελείται προσθέτει τα *jobs* που εκτελούν την ταξινόμηση καθώς και το *job* που θα εκτελεστεί μετά την ολοκλήρωσή της και τερματίζει τη λειτουργία του.
Για τον συχρονισμό της διαδικασίας που θα εκτελεστεί μετα το *sort* χρησιμοποιούμε ένα σεμαφόρο. Ο σεμαφόρος αυξάνεται κάθε φορά που τελειώνει ένα *sort jobs*.Ξέρουμε εξάρχής πόσα *sort jobs* θα εκτελεστούν, οπότε η ουρά του *Job Scheduler* φροντίζει να εξάγει μόνο *jobs* που ο σεμαφόρος τους ισούται με την τιμή που έχει καθοριστεί. Ο αριθμός τον *job* που θα χρησιμοποιηθούν για το *sort* ενός πίνακα καθορίζεται απο την $defined$ μεταβλητή $sort_jobs$.Αυτή η μεταβλητή μπορεί να πάρει τιμές απο 1 μέχρι 256(που είναι το μέγεθοσ του *histogram* του *radix*).

Χρόνοι Υλοποίησης Δοκιμάσαμε να χωρίσουμε την ταξινόμηση σε 1,2,4,16 ίσα *jobs* για να δούμε την επιρροή που έχει αυτή η αλλαγή στο χρόνο εκτέλεσης του προγράμματος.

Για το *medium*:





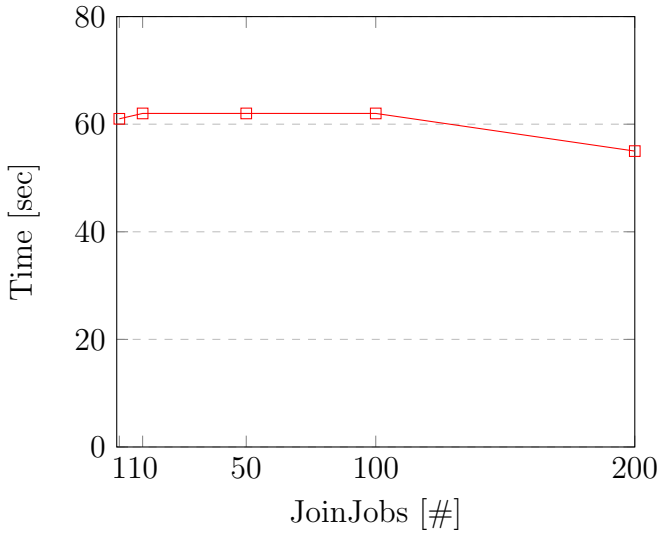
Όπως φαίνεται παραπάνω οι διαφορές δεν είναι πολυ μεγάλες. Παρ΄όλα αυτά η καλύτερη υλοποήση είναι αυτή που χωρίζει το κάθε *sort* σε 2 *jobs*.

2.3.4 Επιπέδου Τελεστή Ζεύξης (Join)

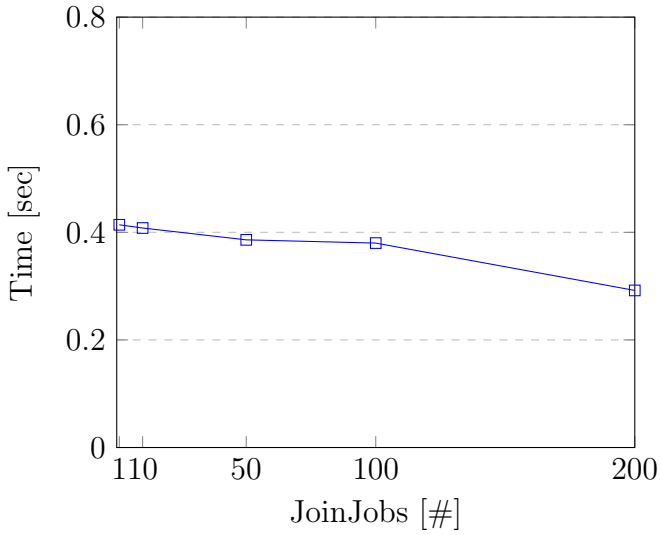
Λεπτομέρειες Υλοποίησης Εκτελούμε τον αλγόριθμο για να κάνουμε *join* μεταξύ δυο ταξινομημενων πινακων και οταν βρούμε δυο στοιχεια που είναι ίσα (δηλαδή θα συμμετέχουν στα αποτελεσματα του *join*) τοτε δημιουργουμε ενα καινουργιο job το οποιο αναλλαμβάνει να διασταυρώσει τα κοινά στοιχεία των δύο πινάκων.
Σε αυτο το σημείο καταφέραμε να μειώσουμε την κατανάλωση μνήμης του προγράμματος, εκτελώντας τα *joins* αφού έχουμε δημιουργήσει όλα τα *jobs* και γνωρίζουμε τον αριθμό των κοινών στοιχείων, με αποτέλεσμα να μην χριαζόμαστε βοηθητική δομή(λίστα). Πιο συγκεκριμένα αυτή η αλλαγή μείωσε την κατανάλωση της μνήμης κατά $1.5MB$
Το κάθε *job* μπορεί να προστείθεται μόνο του ώς ξεχωριστό *job* στον *scheduler* ή να γίνεται ομαδοποίηση ώστε να γίνονται *schedule* λιγότερα *jobs*. Αυτό γίνεται ανάλογα με την $defined$ μεταβλητή $MIN_JOIN_GROUP_ITEMS$ η οποία καθορίζει ποιός είναι ο ελάχιστος αριθμός κοινων στοιχείων που μπορεί να έχει να διασταυρώσει μεταξύ τους ένα *job* που προστίθεται στον *scheduler*.

Χρόνοι Υλοποίησης Ανάλογα με τιμή της $defined$ μεταβλητής $MIN_JOIN_GROUP_ITEMS$ αλλάζει και ο χρόνος εκτέλεσης του προγράμματος όποως φαίνεται στο παρακάτω διάγραμμα.

Για το *medium*:



Για το *small*:



3 Χρόνοι εκτέλεσης του προγράμματος

Στον παρακάτω πίνακα παρουσιάζονται αναλυτικά οι χρόνοι εκτέλεσης του προγράμματος σε κάθε περίπτωση.
Σε αυτές τις εκτελέσεις τα στατιστικά δεν χρισιμοποιήθηκαν, ο αριθμός των *threads* που χτησιμοποιήθηκαν είναι 8, και στο *sort* και το *join* έχουν χρησιμοποιηθεί τα *defines* που έιχαν τον καλύτερο χρόνο.

Query	Join	Sort	Small	Medium
∘	∘	∘	55,260	0,298
∘	∘	χ	57,685	0,303
∘	χ	∘	60,041	0,540
∘	χ	χ	65,368	0,933
χ	∘	∘	56,238	0,303
χ	∘	χ	71	0,312
χ	χ	∘	57,968	0,533
χ	χ	χ	143,349	1,040

4 Επίλογος

Μετά από πολλές δοκιμές διαλέγουμε ως τον καλύτερο τρόπο να τρέξεις το πρόγραμμα με παραλληλοποίηση σε όλα τα επίπεδα, χωρίς την χρήση των στατιστικών, με $MIN_JOIN_GROUP_ITEMS = 200$ και με $SORT_NUM_JOBS = 2$. Οι μετρήσεις αυτές έχουν εκτελεστεί σε μηχάνημα με 16 $GB\ RAM$ και 8 πυρήνες.